Generating MNIST Images with Deep Convolutional Generative Adversarial Networks (DCGANs)

Trianto Haryo Nugroho
Department of Mathematics, Faculty of Mathematics and Natural Sciences, Universitas Indonesia
e-mail: trianto.haryo@ ui.ac.id

ABSTRACT

In recent years, convolutional networks (CNNs) have been widely used for supervised learning tasks in computer vision, but their application in unsupervised learning remains less explored. This work focuses on generating MNIST images using Deep Convolutional Generative Adversarial Networks (DCGANs), a specialized class of CNNs with architectural constraints designed for generative modeling. By training DCGANs on the MNIST dataset, we demonstrate their ability to learn hierarchical representations, from basic digit strokes to complete digit structures, in both the generator and discriminator. Additionally, we highlight the utility of the learned features for various tasks, showcasing their potential as effective image representations in unsupervised learning contexts.

Keywords: *DCGAN, Deep Convolutional Generative Adversarial Networks, MNIST, Generative Modeling, Unsupervised Learning, Convolutional Neural Networks, Image Generation, Hierarchical Representations, Generator, Discriminator*

I. INTRODUCTION

Learning effective feature representations from large-scale unlabeled datasets is a prominent area of research. In the field of computer vision, the abundance of unlabeled images and videos presents an opportunity to develop robust intermediate representations that can be applied to supervised tasks like image classification. This paper explores the use of Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) to generate MNIST and FashionMNIST images and leverage the generator and discriminator networks as feature extractors for downstream tasks. Unlike traditional maximum likelihood approaches, GANs offer an appealing framework due to their unique learning dynamics and the absence of heuristic cost functions, such as pixel-wise mean squared error. Despite their potential, GANs are notoriously challenging to train, often leading to unstable models or incoherent outputs. Furthermore, there has been limited research on understanding and visualizing the learned representations within GANs, particularly in the context of multi-layer architectures. This study addresses these gaps by demonstrating the utility of GANs in generating MNIST and FashionMNIST images and uncovering their learned hierarchical representations.

In this paper, we make the following key contributions:

- We propose and evaluate a set of architectural constraints for Convolutional GANs, which enhance their stability during training across various scenarios. We refer to this class of architectures as Deep Convolutional GANs (DCGANs).
- We demonstrate the use of trained discriminators for image classification tasks, achieving competitive performance compared to other unsupervised learning methods.
- We visualize the filters learned by GANs and show empirically that specific filters specialize in generating specific objects.
- We highlight the generators' intriguing vector arithmetic properties, which enable

straightforward manipulation of various semantic attributes in the generated MNIST and FashionMNIST samples.

II. RELATED WORK

2.1 REPRESENTATION LEARNING FROM UNLABELED DATA

Unsupervised representation learning is a well-researched topic in computer vision, particularly in the context of images. A traditional approach to unsupervised representation learning involves clustering the data (e.g., using K-means) and using the resulting clusters to improve classification performance. In image processing, hierarchical clustering of image patches (Coates & Ng, 2012) is often employed to learn robust image representations. Another widely used technique is training autoencoders, including convolutional autoencoders, stacked autoencoders (Vincent et al., 2010), and those that separate the "what" and "where" components of the code (Zhao et al., 2015), as well as ladder networks (Rasmus et al., 2015). These models encode an image into a compact representation and then decode it to reconstruct the image as accurately as possible. These methods have proven effective in learning meaningful feature representations from image pixels. Additionally, deep belief networks (Lee et al., 2009) have also been successful in learning hierarchical representations.

2.2 GENERATING NATURAL IMAGES

Generative image models have been widely studied and can be categorized into two types: parametric and non-parametric. Non-parametric models typically generate images by matching patterns from a database of existing images, often using patch-based matching techniques. These models have been successfully applied in areas such as texture synthesis (Efros et al., 1999), super-resolution (Freeman et al., 2002), and in-painting (Hays & Efros, 2007).

A. Parametric models

Parametric models, on the other hand, have been extensively explored for generating specific types of images, such as MNIST digits or textures (Portilla & Simoncelli, 2000). However, generating realistic, natural images of the real world has remained a challenging task until recent advancements. Variational sampling methods (Kingma & Welling, 2013) have shown promise but often produce blurry results. Another method involves an iterative forward diffusion process (Sohl-Dickstein et al., 2015) to generate images.

B. Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) initially struggled with producing noisy and incoherent images. A laplacian pyramid extension (Denton et al., 2015) improved image quality but introduced wobbliness in objects due to noise from combining multiple models. Recent approaches, such as recurrent networks

(Gregor et al., 2015) and deconvolutional networks (Dosovitskiy et al., 2014), have also achieved some success in generating natural images. However, these methods have not utilized the generators for downstream supervised tasks.

C. Images Dataset

The MNIST dataset is a widely used benchmark in machine learning, consisting of 60,000 training images and 10,000 test images of handwritten digits from 0 to 9, each represented as a grayscale 28x28 pixel image. It serves as a foundational dataset for evaluating classification models. Fashion MNIST, on the other hand, is a more challenging alternative, designed to address the simplicity of MNIST by providing 28x28 grayscale images of 10 clothing categories, such as T-shirts, trousers, and shoes. Both datasets are structured similarly, enabling seamless testing of algorithms, with Fashion MNIST offering a closer approximation to real-world image recognition challenges due to its higher variability and complexity. An example image from the MNIST and Fashion MNIST datasets is shown in Fig. 1 below.

MNIST Dataset Fashion MNIST Dataset

Fig 1. MNIST and Fashion MNIST Dataset

2.3 VISUALIZING THE INTERNALS OF CNNS

A common criticism of neural networks is that they operate as "black-box" models, with limited understanding of their internal workings in terms of simple, human-understandable algorithms. In the case of Convolutional Neural Networks (CNNs), Zeiler et al. (Zeiler & Fergus, 2014) demonstrated that by applying deconvolutions and highlighting the maximal activations, it is possible to approximate the function of each convolution filter in the network. Similarly, using gradient descent on the inputs allows us to examine the ideal image that activates specific subsets of filters (Mordvintsev et al.).

III. PROBLEM DEFINITIONS

The primary challenge stems from the limitations of conventional generative models and standard GANs in generating high-quality, realistic images of natural scenes. Traditional generative models often produce blurry or noisy images and fail to fully capitalize on their potential for supervised learning tasks. On the other hand, standard GANs, despite their theoretical appeal, frequently generate images that are unrealistic and noisy due to the adversarial training process. Deep Convolutional Generative Adversarial Networks (DCGANs) were developed to address these shortcomings by

incorporating convolutional layers into both the generator and discriminator networks. This approach aims to improve image quality by tackling issues like blurriness and noise, enhance the stability of GAN training for consistent production of high-quality images, and create a framework that effectively applies generative models in supervised learning tasks. The goal is to generate sharper, more realistic images while maximizing the applicability of generative models across a range of tasks.

Moreover, DCGANs contribute to overcoming the instability in the training of standard GANs by introducing modifications such as replacing pooling layers with strided convolutions and using batch normalization. These adjustments help ensure smoother training dynamics and more reliable convergence. Additionally, the use of convolutional layers allows the networks to learn spatial hierarchies, which is crucial for generating high-quality images of complex natural scenes. This ability to capture fine-grained spatial features makes DCGANs particularly suited for tasks that require high resolution and realism, such as image synthesis, super-resolution, and style transfer.

Furthermore, DCGANs hold significant promise for extending the capabilities of generative models in practical applications, such as data augmentation and transfer learning. By producing realistic images that resemble real-world data distributions, they can be used to enhance supervised learning models by supplementing limited datasets with synthetic examples. This capability makes DCGANs a powerful tool in fields like medical imaging, where labeled data is scarce, or in creative industries like art and design, where generating novel content is highly valuable. Thus, DCGANs represent a significant step forward in making generative models more practical and beneficial for a wide range of real-world applications.

IV. APPROACH AND MODEL ARCHITECTURE

Previous attempts to scale GANs with CNNs for modeling images were unsuccessful, which led the creators of LAPGAN (Denton et al., 2015) to develop an alternative method of iteratively upscaling low-resolution generated images, making them easier to model reliably. We faced similar challenges when trying to scale GANs using traditional CNN architectures from supervised learning. However, after extensive model exploration, we identified a set of architectures that led to stable training across various datasets and enabled the training of higher-resolution and deeper generative models.

Our approach is centered around modifying three key changes to CNN architectures. The first is adopting the all-convolutional network (Springenberg et al., 2014), which replaces conventional spatial pooling methods (like maxpooling) with strided convolutions, allowing the network to learn its own downsampling. We applied this approach in both our generator (to learn spatial upsampling) and discriminator.

The second change is the trend of removing fully connected layers after convolutional features. A prime example is global average pooling, which has been used in state-of-the-art image classification models (Mordvintsev et al.). We found that global average pooling enhanced model stability, though it slowed convergence. A compromise was reached by directly connecting the highest convolutional features to the input and output of the generator and discriminator. The first GAN layer, which takes a uniform noise distribution Z as input, may be considered fully connected since it performs a matrix

multiplication. However, the result is reshaped into a 4D tensor, which then begins the convolution stack. For the discriminator, the final convolution layer is flattened and passed into a single sigmoid output.

The third modification involves Batch Normalization (Ioffe & Szegedy, 2015), which normalizes inputs to each unit to have zero mean and unit variance, stabilizing learning. This addresses issues related to poor initialization and improves gradient flow in deeper models. Batch normalization was critical in preventing the generator from collapsing to a single point, a common issue in GANs. However, applying batch normalization to all layers resulted in sample oscillation and instability, which we mitigated by excluding batch normalization from the generator's output layer and the discriminator's input layer.

We used the ReLU activation (Nair & Hinton, 2010) in the generator, except for the output layer, which uses the Tanh function. We observed that the Tanh activation helped the model learn more quickly and cover the color space of the training distribution. In contrast, the discriminator used the leaky rectified linear unit (Maas et al., 2013) (Xu et al., 2015), which worked particularly well for higher-resolution models. This differs from the original GAN paper, which employed the maxout activation (Goodfellow et al., 2013).

Architecture guidelines for ensuring stable Deep Convolutional GANs include:

- Substitute pooling layers with strided convolutions in the discriminator and fractional-strided convolutions in the generator.
- Implement batch normalization in both the generator and discriminator.
- Eliminate fully connected hidden layers in deeper architectures.
- Apply ReLU activation in the generator for all layers except the output layer, which should use Tanh.
- Use LeakyReLU activation in all layers of the discriminator.

IV. EXPERIMENT

This section will describe the code and the purpose of each function, specifically for the implementation using the DCGAN model developed by (Radford et al., 2015).

4.1 IMPORT LIBRARY

This code snippet in Fig. 2 imports the necessary components for building a deep learning model using Keras and TensorFlow. The main libraries are related to defining layers for the neural network, optimizing the model, and loading the MNIST dataset. It sets the stage for building a deep convolutional network (likely a DCGAN, as mentioned earlier) by preparing the layers required for both the generator and discriminator in the GAN architecture. The additional libraries like NumPy, PIL, argparse, and math will help with data manipulation, image handling, and the configuration of model parameters.

```
from keras.models import Sequential
from keras.layers import Dense, Reshape
from tensorflow.keras.layers import Activation, BatchNormalization, UpSampling2D, Dense, Reshape, Conv2D, MaxPooling2D, Flatten
from keras.optimizers import SGD
from keras.datasets import mnist
import numpy as np
from PIL import Image
import argparse
import math
```

Fig 2. Import Library

4.2 GENERATOR MODEL FUNCTION

The generator_model() function in Fig. 3 defines a generator model for a GAN that generates images from a random noise vector. The model starts with a Dense layer to transform the input into a high-dimensional representation, followed by BatchNormalization and a tanh activation function to stabilize training. It then uses Reshape to convert the output into a 7x7 tensor with 128 channels, and two UpSampling2D layers to double the spatial size of the image. Next, Conv2D layers are applied to perform convolutions and refine the image details, producing a final output of size 28x28 with 1 channel (grayscale). The final tanh activation ensures pixel values are in the range of -1 to 1.

```
def generator_model():
  model = Sequential()
  model.add(Dense(units=1024, input_dim=100))
  model.add(Activation('tanh'))
  model.add(Dense(128*7*7))
  model.add(BatchNormalization())
  model.add(Activation('tanh'))
  model.add(Reshape((7, 7, 128), input_shape=(128*7*7,)))
  model.add(UpSampling2D(size=(2, 2)))
  model.add(Conv2D(64, (5, 5), padding='same'))
  model.add(Activation('tanh'))
  model.add(UpSampling2D(size=(2, 2)))
  model.add(Conv2D(1, (5, 5), padding='same'))
  model.add(Activation('tanh'))
  return model
```

Fig 3. Generator Model Function

4.3 DISCRIMINATOR MODEL FUNCTION

The discriminator_model() function in Fig. 4 defines the discriminator model for a GAN, which classifies images as real or fake. The model begins with a Conv2D layer that processes the 28x28 grayscale input image, followed by a tanh activation function. MaxPooling2D layers are then applied to downsample the image and extract important features. The model includes another Conv2D layer with increased filters (128) for further feature extraction, followed by another MaxPooling2D layer to reduce spatial dimensions. The output is then flattened, and two Dense layers are added, with a tanh activation in the first and a sigmoid activation in the final layer. The sigmoid output produces a probability value between 0 and 1, indicating whether the input image is real or fake.

```
def discriminator_model():
  model = Sequential()
  model.add(Conv2D(64, (5, 5), padding='same', input_shape=(28, 28, 1))
  model.add(Activation('tanh'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Conv2D(128, (5, 5)))
  model.add(Activation('tanh'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(1024))
  model.add(Activation('tanh'))
  model.add(Dense(1))
  model.add(Activation('sigmoid'))
  return model
```

Fig 4. Discriminator Model Function

4.4 GENERATOR CONTAINING DISCRIMINATOR FUNCTION

The generator_containing_discriminator() function in Fig. 5 defines a model that combines the generator and discriminator for a GAN. The function takes in two models: the generator (g) and the discriminator (d). The generator is added to the sequential model first, which generates images from random noise. The discriminator's trainable attribute is set to False to prevent it from being updated during the generator's training. After the generator, the discriminator is added to the model to classify the generated images as real or fake. This combined model allows for training the generator to improve its image generation, while the discriminator remains fixed during this phase.

```
def generator_containing_discriminator(g, d):
  model = Sequential()
  model.add(g)
  d.trainable = False
  model.add(d)
  return model
```

Fig 5. Generator Containing Discriminator Function

4.5 COMBINES IMAGE FUNCTION

The combine_images() function in Fig. 6 takes in a batch of generated images and arranges them into a single grid for visualization. The function first calculates the number of rows (height) and columns (width) for the grid based on the number of images to display. It then initializes a blank image array with the appropriate dimensions to hold all the images. For each image in the batch, the function places it into the corresponding position in the grid by calculating the row (i) and column (j) indices based on the current image index. The resulting grid image is returned, where each generated image is placed in its respective position within the grid.

```python
def combine_images(generated_images):
  num = generated_images.shape[0]
  width = int(math.sqrt(num))
  height = int(math.ceil(float(num)/width))
  shape = generated_images.shape[1:3]
  image = np.zeros((height*shape[0], width*shape[1]), dtype=generated_images.dtype)
  for index, img in enumerate(generated_images):
    i = int(index/width)
    j = index % width
    image[i*shape[0]:(i+1)*shape[0], j*shape[1]:(j+1)*shape[1]]= img[:, :, 0]
  return image
```

Fig 6. Combines Image Function

4.6 TRAINING FUNCTION

The train() function in Fig. 7 defines the training process for a GAN with a generator and discriminator model. First, it loads and preprocesses the MNIST dataset by scaling the pixel values to the range of -1 to 1 and reshaping the data to fit the model. The discriminator and generator models are created using the discriminator_model() and generator_model() functions, and the combined model (generator + discriminator) is also defined. The function then sets up the optimizers for both models using Stochastic Gradient Descent (SGD) with specific learning rates and momentum. The models are compiled with binary cross-entropy loss and appropriate optimizers.

In the training loop, the function iterates over 20 epochs, and for each epoch, it processes the training data in batches. For each batch, the function generates random noise and uses the generator to create fake images, which are combined with real images from the training data. The discriminator is trained on both real and generated images, and its loss is computed. After that, the generator is trained by freezing the discriminator's weights, ensuring the generator learns to fool the discriminator. Every 20 batches, the generated images are saved for visualization, and every 10 batches, the weights of both the generator and discriminator are saved to disk. This process continues until all epochs are completed, with the generator learning to produce increasingly realistic images over time.

```python
def train(BATCH_SIZE):
    (X_train, y_train), (X_test, y_test) = mnist.load_data()
    X_train = (X_train.astype(np.float32) - 127.5)/127.5
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)

    # Create discriminator and generator models
    d = discriminator_model()
    g = generator_model()
    d_on_g = generator_containing_discriminator(g, d)

    # Optimizers for generator and discriminator
    d_optim = SGD(learning_rate=0.0005, momentum=0.9, nesterov=True)
    g_optim = SGD(learning_rate=0.0005, momentum=0.9, nesterov=True)

    g.compile(loss='binary_crossentropy', optimizer=g_optim)
    d.trainable = True
    d.compile(loss='binary_crossentropy', optimizer=d_optim)
    d_on_g.compile(loss='binary_crossentropy', optimizer=g_optim)

    # Training loop
    for epoch in range(20):
        print(f"Epoch is {epoch}")
        for index in range(int(X_train.shape[0]/BATCH_SIZE)):
            # Generate noise for the generator
            noise = np.random.uniform(-1, 1, size=(BATCH_SIZE, 100))

            # Get a batch of real images
            image_batch = X_train[index*BATCH_SIZE:(index+1)*BATCH_SIZE]

            # Generate images from the noise
            generated_images = g.predict(noise, verbose=0)

            # Save generated images every 20 batches
            if index % 20 == 0:
                image = combine_images(generated_images)
                image = image*127.5+127.5
                Image.fromarray(image.astype(np.uint8)).save(f"{epoch}_{index}.png")

            # Train discriminator
            X = np.concatenate((image_batch, generated_images))
            y = np.concatenate((np.ones(BATCH_SIZE), np.zeros(BATCH_SIZE)))
            d_loss = d.train_on_batch(X, y)
            print(f"Batch {index} d_loss: {d_loss}")

            # Train generator
            noise = np.random.uniform(-1, 1, size=(BATCH_SIZE, 100))
            d.trainable = False
            g_loss = d_on_g.train_on_batch(noise, np.ones(BATCH_SIZE))
            d.trainable = True
```

Fig 7. Training Function

4.7 IMAGE GENERATOR FUNCTION

The generate() function in Fig. 8 generates images using the trained generator model. It first initializes the generator model using the generator_model() function and loads the pre-trained weights from the saved generator file. If the nice parameter is set to True, it also initializes the discriminator model and loads the pre-trained weights for the discriminator. The function then generates a batch of random noise vectors and uses the generator to create corresponding images. The discriminator is then used to predict the quality of these generated images. The generated images are sorted based on the discriminator's predictions, and the top images are selected to form a batch of "nice" images. These images are then combined into a single image using the combine_images() function.

If the nice parameter is not specified, the function generates a batch of images without considering the discriminator's feedback. In this case, it simply creates random noise, generates images using the generator, and combines them into a single image. Regardless of the method, the generated image is rescaled back to the pixel value range of 0 to 255, and it is saved as a PNG file named "generated_image.png". This function allows for both regular and discriminator-guided image generation, providing flexibility in producing high-quality generated images based on the model's performance.

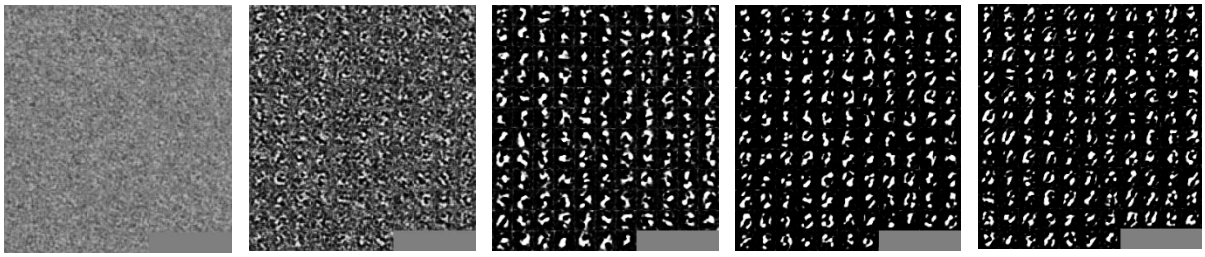
```
def generate(BATCH_SIZE, nice=False):
  g = generator_model()
  g.compile(loss='binary_crossentropy', optimizer="SGD")
  g.load_weights('generator')
  if nice:
    d = discriminator_model()
    d.compile(loss='binary_crossentropy', optimizer="SGD")
    d.load_weights('discriminator')
    noise = np.uniform(-1, 1, (BATCH_SIZE*20, 100))
    generated_images = g.predict(noise, verbose=1)
    d_pred = d.predict(generated_images, verbose=1)
    index = np.arange(0, BATCH_SIZE*20)
    index.resize((BATCH_SIZE*20, 1))
    pre_with_index = list(np.append(d_pred, index, axis=1))
    pre_with_index.sort(key=lambda x: x[0], reverse=True)
    nice_images = np.zeros((BATCH_SIZE,) + generated_images.shape[1:3], dtype=np.float32)
    nice_images = nice_images[:, :, :, None]
    for i in range(BATCH_SIZE):
      idx = int(pre_with_index[1][1])
      nice_images[i, :, :, 0] = generated_images[idx, :, :, 0]
    image = combine_images(nice_images)
  else:
    noise = np.random.uniform(-1, 1, (BATCH_SIZE, 100))
    generated_images = g.predict(noise, verbose=1)
    image = combine_images(generated_images)
  image = image*127.5+127.5
  Image.fromarray(image.astype(np.uint8)).save("generated_image.png")
```

Fig 8. Image Generator Function

V. RESULT AND DISCUSSION

5.1 GENERATED MNIST IMAGES

Fig. 9 shows the generated MNIST images at epoch 0, illustrating the progress of the generator in producing images from random noise. The images presented correspond to different batches (0, 200, 300, 400, and 460) during the first epoch of training. Initially, the images appear blurry and lack distinct features, which is typical for early stages of training when the generator has not yet learned the patterns and structure of the MNIST digits. As the training progresses, the quality of the generated images will improve, with clearer and more recognizable digit representations emerging as the generator refines its ability to produce images that resemble the real MNIST dataset. The varying quality of the generated images at different batches reflects the gradual learning and adjustment of the generator model throughout the epoch.



epoch 0 batch 0 | epoch 0 batch 200 | epoch 0 batch 300 | epoch 0 batch 400 | epoch 0 batch 460

Fig 9. Generated MNIST Images of epoch 0

Fig. 10 shows the generated MNIST images at epoch 1, providing an insight into the improvement of the generator's output as training progresses. The images shown correspond to different batches (0, 100, 200, 300, and 340) in the second epoch. Compared to epoch 0, the images at this stage begin to exhibit more distinct shapes and clearer outlines of digits, indicating that the generator has started to learn the structure of MNIST digits. Although the generated images are still not perfect, the shapes are more recognizable and less blurry, reflecting the model's progress in refining its ability to generate realistic images. As the generator continues to train, the quality of the images will likely improve further, with more defined and accurate representations of digits. The varying quality between batches highlights the ongoing adjustments and fine-tuning of the model.

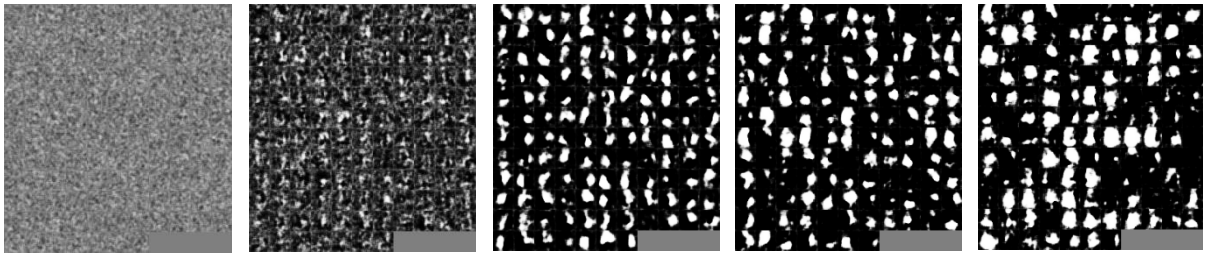
epoch 1 batch 0 | epoch 1 batch 100 | epoch 1 batch 200 | epoch 1 batch 300 | epoch 1 batch 340

Fig 10. Generated MNIST Images of epoch 1

5.2 GENERATED FASHION MNIST IMAGES

Fig. 11 shows the generated Fashion MNIST images at epoch 0, providing a glimpse of the generator's initial output during training. The images correspond to different batches (0, 200, 300, 400, and 460) in the first epoch. At this early stage, the generated images appear blurry and lack clear details, with the shapes of clothing items such as shirts, shoes, and trousers still indistinct. The generator has yet to fully learn the underlying patterns of the Fashion MNIST dataset, resulting in images that are rough and hard to interpret. Despite the lack of clarity, these initial outputs serve as a foundation for further training, where the generator will refine its ability to produce more realistic and recognizable images of fashion items in subsequent epochs. The variety of images across different batches highlights the

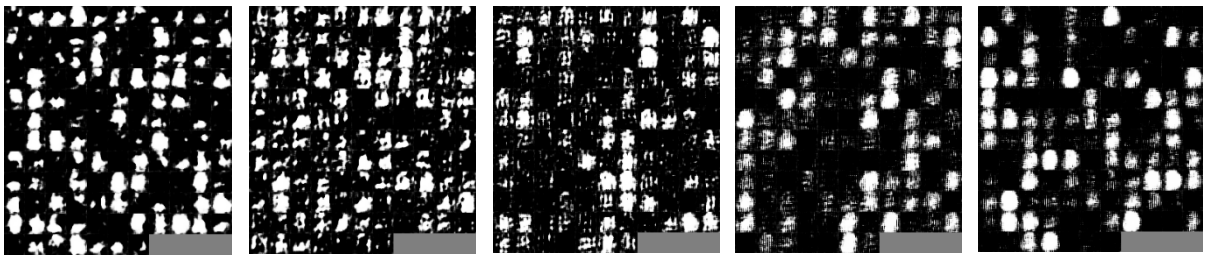
diverse, but still underdeveloped, attempts by the generator at creating fashion-related images.



epoch 0 batch 0	epoch 0 batch 200	epoch 0 batch 300	epoch 0 batch 400	epoch 0 batch 460

Fig 11. Generated Fashion MNIST Images of epoch 0

Fig. 12 shows the generated Fashion MNIST images at epoch 1, showcasing the progress of the generator during its second round of training. The images from different batches (0, 200, 300, and 360) exhibit slight improvements compared to the previous epoch, with more recognizable features such as clearer outlines of clothing items like shirts, shoes, and trousers. However, the images still lack fine details and may appear somewhat blurry or distorted. These outputs demonstrate that the generator is gradually learning the structure and patterns of the Fashion MNIST dataset, but the training process is still in its early stages. As the training continues, the images will likely become sharper and more realistic, with greater accuracy in representing various fashion items. The variety of images across different batches reflects the generator's ongoing exploration of the dataset's latent space.



epoch 1 batch 0	epoch 1 batch 200	epoch 1 batch 200	epoch 1 batch 300	epoch 1 batch 360

Fig 12. Generated Fashion MNIST Images of epoch 1

5.3 MODEL PERFORMANCE

Based on the training results for epoch 0, the loss values for both the discriminator (d_loss) and generator (g_loss) show consistent decreases as the batches progress. Initially, d_loss starts at approximately 0.69 and gradually decreases to around 0.49 by the final batch. Similarly, g_loss follows the same pattern, decreasing from around 0.69 to 0.49. This indicates that the model is learning effectively, with the generator improving its ability to produce data resembling the real data, while the discriminator continues to adapt to distinguish real data from generated data. However, it is important to monitor for potential issues such as mode collapse or an imbalance between the discriminator and generator in future iterations.

The results in Fig. 13 indicate that the generator and discriminator are well-balanced, with loss values converging close to 0.50 by the final batch. This balance suggests that the generator produces data that is increasingly realistic, while the discriminator effectively differentiates between real and generated data without overpowering the generator. Achieving a loss near 0.50 is a good indicator of stable training, as it reflects a healthy competition between the two components of the GAN model.

```
Epoch is 1
Batch 0 d_loss: 0.43365657329559326
Batch 0 g_loss: [array(0.43365657, dtype=float32), array(0.43365657, dtype=float32), array(0.43365657, dtype=float32)]
Batch 1 d_loss: 0.43379044532775888
Batch 1 g_loss: [array(0.43379045, dtype=float32), array(0.43379045, dtype=float32), array(0.43379045, dtype=float32)]
Batch 2 d_loss: 0.4339394271373749
Batch 2 g_loss: [array(0.43393943, dtype=float32), array(0.43393943, dtype=float32), array(0.43393943, dtype=float32)]
Batch 3 d_loss: 0.43418753147125244
Batch 3 g_loss: [array(0.43418753, dtype=float32), array(0.43418753, dtype=float32), array(0.43418753, dtype=float32)]
Batch 4 d_loss: 0.43436098098754883
Batch 4 g_loss: [array(0.43436098, dtype=float32), array(0.43436098, dtype=float32), array(0.43436098, dtype=float32)]
Batch 5 d_loss: 0.4346216022968292
Batch 5 g_loss: [array(0.4346216, dtype=float32), array(0.4346216, dtype=float32), array(0.4346216, dtype=float32)]
Batch 6 d_loss: 0.43482738733291626
Batch 6 g_loss: [array(0.4348274, dtype=float32), array(0.4348274, dtype=float32), array(0.4348274, dtype=float32)]
Batch 7 d_loss: 0.4350474178791046
Batch 7 g_loss: [array(0.43504742, dtype=float32), array(0.43504742, dtype=float32), array(0.43504742, dtype=float32)]
Batch 8 d_loss: 0.4352295994758606
Batch 8 g_loss: [array(0.4352296, dtype=float32), array(0.4352296, dtype=float32), array(0.4352296, dtype=float32)]
Batch 9 d_loss: 0.435411274433136
Batch 9 g_loss: [array(0.43541127, dtype=float32), array(0.43541127, dtype=float32), array(0.43541127, dtype=float32)]
Batch 10 d_loss: 0.4355551600456238
Batch 10 g_loss: [array(0.43555516, dtype=float32), array(0.43555516, dtype=float32), array(0.43555516, dtype=float32)
Batch 11 d_loss: 0.43584010004997253
Batch 11 g_loss: [array(0.4358401, dtype=float32), array(0.4358401, dtype=float32), array(0.4358401, dtype=float32)]
Batch 12 d_loss: 0.4360259175300598
Batch 12 g_loss: [array(0.43602592, dtype=float32), array(0.43602592, dtype=float32), array(0.43602592, dtype=float32)
Batch 13 d_loss: 0.43611109256744385
Batch 13 g_loss: [array(0.4361111, dtype=float32), array(0.4361111, dtype=float32), array(0.4361111, dtype=float32)]
Batch 14 d_loss: 0.4363189935684204
Batch 14 g_loss: [array(0.436319, dtype=float32), array(0.436319, dtype=float32), array(0.436319, dtype=float32)]
Batch 15 d_loss: 0.4364578425884247
Batch 15 g_loss: [array(0.43645784, dtype=float32), array(0.43645784, dtype=float32), array(0.43645784, dtype=float32)
Batch 16 d_loss: 0.4366190731525421
Batch 16 g_loss: [array(0.43661907, dtype=float32), array(0.43661907, dtype=float32), array(0.43661907, dtype=float32)
Batch 17 d_loss: 0.4366990625858307
Batch 17 g_loss: [array(0.43669906, dtype=float32), array(0.43669906, dtype=float32), array(0.43669906, dtype=float32)
Batch 18 d_loss: 0.43691301345825195
Batch 18 g_loss: [array(0.436913, dtype=float32), array(0.436913, dtype=float32), array(0.436913, dtype=float32)]
```

Fig 13. Model Performance of epoch 1

One limitation of training the model using Google Colab as shown in Fig. 14, even with the T4 runtime, is the constrained resources, particularly RAM and need to upgrade to Colab Pro. In this case, the training process could only run for 2 epochs before the runtime was interrupted due to excessive RAM usage, despite the need for at least 100 epochs to achieve optimal results. Training to 100 epochs is essential to generate images that closely resemble the original and to observe progressively decreasing loss values, indicating improved model performance. The inability to complete the required epochs limits the potential of the model and the quality of the generated outputs, highlighting the need for more robust computational resources.

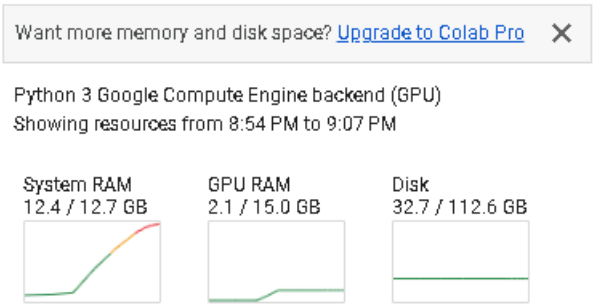


Fig 14. RAM Usage Limitations

VI. CONCLUSION

From the experiments conducted on DCGANs for generating MNIST and Fashion MNIST images, several conclusions can be drawn:

- The DCGAN framework successfully overcomes key limitations of traditional generative models, leveraging convolutional layers and adversarial training to produce higher-quality and more realistic images.
- The progression of image quality over epochs underscores the critical role of iterative training and the dynamic interaction between the generator and discriminator. The choice of activation functions further influences the visual characteristics of the generated outputs.
- DCGANs demonstrate considerable potential for applications that demand realistic image generation. Their capacity for continuous improvement with training suggests a promising future for advancements in both unsupervised and supervised learning contexts
- The experiments revealed limitations in computational resources, particularly when using platforms like Google Colab. Even with high-performance runtimes, RAM constraints restricted training to only two epochs instead of the required 100 epochs for optimal results.
- Limited training epochs affected the ability to generate highly realistic images and achieve significant convergence in loss values.
- Generating high-quality outputs requires robust computational resources capable of supporting extended training durations, such as access to higher RAM capacities or premium services like Colab Pro.
- Despite the challenges, the experiments confirm the effectiveness of DCGANs in generating realistic images and their significant potential for broader applications in generative modeling tasks.

VII. FURTHER RESEARCH SUGGESTION

To improve the quality of image generation with DCGAN, the following suggestions can be implemented to boost the model's performance:

- Stabilizing Training Architectures: Future research can focus on refining more stable architectures for training generative adversarial networks (GANs). While the current models achieve impressive results, occasional instability, such as the collapse of certain

filters into a single oscillating mode during longer training sessions, needs to be addressed. Investigating ways to prevent such instabilities will contribute to more robust model performance in the long term.

- Improved Regularization Techniques: Incorporating regularization techniques such as dropout in the discriminator could be explored further. By randomly dropping units during training, dropout forces the network to generalize better, preventing overfitting and enabling the model to learn more robust features. Experimenting with different forms of regularization could improve the model's performance.
- Enhancing Training Stability with Weight Normalization: Weight normalization could be applied to network parameters to help stabilize the training process. This adjustment might speed up convergence and lead to better overall performance by improving the stability of the network's training dynamics.
- Optimization of Hyperparameters: Further investigation into the optimization of hyperparameters—such as learning rates, batch size, and number of epochs—could significantly impact the performance of GANs. Experimenting with different values and optimizing them dynamically during training could enhance the model's ability to generate high-quality images more efficiently.
- Transitioning to Advanced Optimizers: Switching from traditional optimization techniques like SGD to the Adam optimizer could help improve training stability and convergence speed. Adam adapts the learning rate for each parameter individually, making it particularly well-suited for training deep networks and might lead to more reliable results in generating high-quality images.
- Spectral Normalization for Discriminator Layers: Implementing spectral normalization in the discriminator's layers could help further stabilize the training process. This technique could improve the discriminator's ability to distinguish between real and generated images, thereby contributing to a more stable and reliable GAN model.
- Incorporating Label Information: Future research could focus on improving the model's capacity to generate more diverse and complex images by incorporating additional features or altering the input representation. For instance, adding class labels to the GAN model could help the generator create images that are more aligned with specific categories, enabling the model to capture more intricate structures within the data.
- Exploring Latent Space Properties: Further investigations into the properties of the learned latent space could provide valuable insights into how the generator represents complex visual features. Exploring how latent vectors interact and how these relationships can be manipulated could lead to better control over the generated outputs.
- Expanding DCGAN Applications: Extending the application of DCGANs to other domains such as video generation (for frame prediction) and audio generation (for tasks like speech synthesis) could be an interesting area for future research. These expansions would challenge the framework to adapt to more dynamic data types and could provide new opportunities for generating highly realistic outputs in diverse contexts.
- Addressing Visual Misalignment: Future work can aim to address issues related to visual misalignment in the generated images, especially when performing arithmetic operations in the latent space. The misalignment of vectors when applying arithmetic transformations could result in noisy outputs, and methods to better align these transformations should be explored.
- DCGANs (Deep Convolutional Generative Adversarial Networks) have proven highly

effective in generating realistic images, showcasing their potential for a variety of applications:

- Image Generation: DCGANs are widely used for generating high-quality, realistic images. These applications span across various domains such as fashion, art, and design, where the network learns to generate novel images from a distribution of real data.
- Data Augmentation: DCGANs can be used to augment datasets, particularly in scenarios where collecting large amounts of real data is difficult or expensive. By generating additional realistic images, they can help improve the performance of supervised learning models.
- Super-Resolution: DCGANs can be used for enhancing the resolution of low-quality images, allowing for more detailed and clearer outputs, which has applications in fields like medical imaging and satellite imagery.
- Style Transfer: The ability of DCGANs to learn complex distributions of image features makes them useful in style transfer tasks, where the model generates images that combine the content of one image with the style of another.
- Creative Design: DCGANs have also been employed in artistic fields to generate unique and creative designs. Artists and designers use DCGANs to create novel patterns, artworks, or concept designs for fashion, interior design, and advertising.
- Facial Recognition: With datasets like the Faces dataset used in this study, DCGANs can help enhance facial recognition systems by generating synthetic faces for training. This can be particularly useful in improving models when labeled data is limited.

- To create a comparison of generated images using a traditional GAN and a DCGAN, as well as other advanced GAN methods, similar to Fig. 15, you would typically follow these steps to create and organize the visual results:

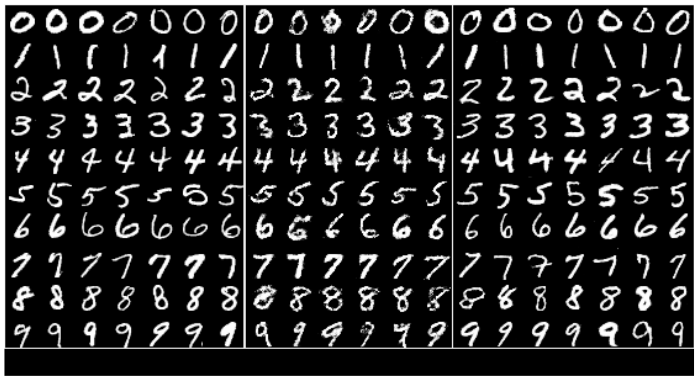


Fig 15. Side-by-side illustration of (from left-to-right) the MNIST dataset, generations from a baseline GAN, and generations from our DCGAN

- To generate color images like the ones in Fig. 16: Generated Bedrooms with DCGAN, you'll need to follow a slightly different approach than for grayscale images like MNIST. Specifically, you need to use a DCGAN trained on a color image dataset, such as the LSUN Bedroom dataset, which contains high-resolution bedroom images.

Fig 16. Generated Bedrooms with DCGAN

- To generate faces using Deep Convolutional Generative Adversarial Networks (DCGAN), where the generated images are combinations of multiple existing faces (as shown in Fig. 17: Generated Faces with DCGAN), you would follow a few steps involving training the DCGAN on a face dataset and then using the trained model to generate faces that may represent combinations of the input data. Here's how you can proceed:

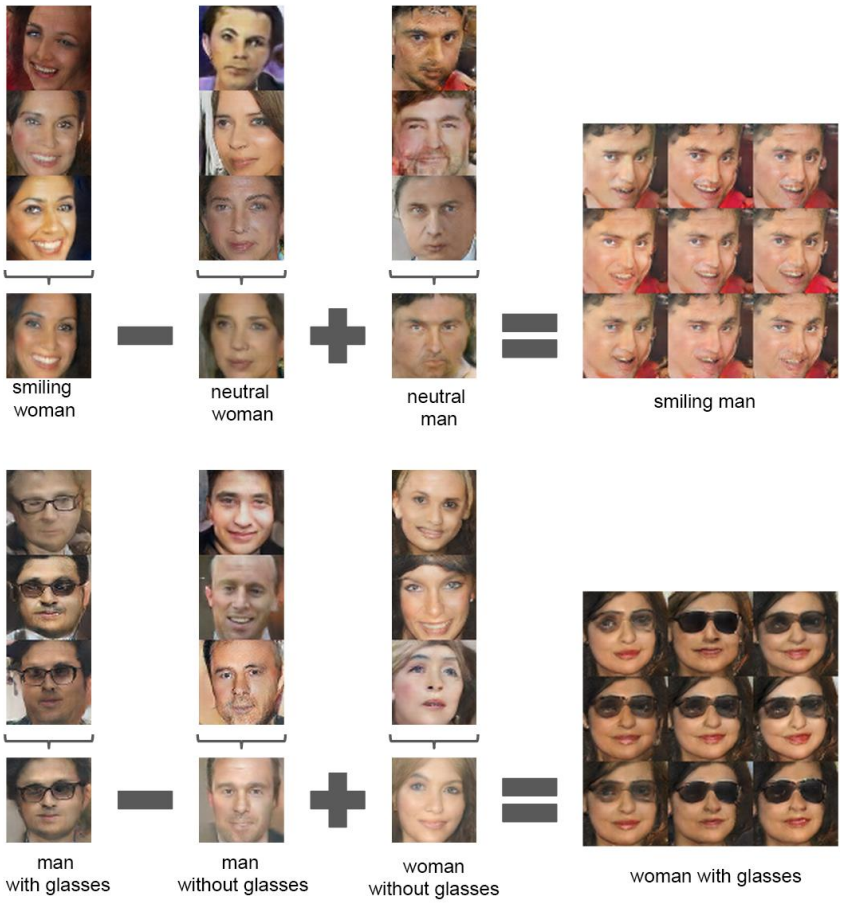


Fig 17. Generated Faces with DCGAN

REFERENCES

Bergstra, James and Bengio, Yoshua. Random search for hyper-parameter optimization. JMLR, 2012.

Coates, Adam and Ng, Andrew. Selecting receptive fields in deep networks. NIPS, 2011.

Coates, Adam and Ng, Andrew Y. Learning feature representations with k-means. In Neural Networks: Tricks of the Trade, pp. 561–580. Springer, 2012.

Deng, Jia, Dong, Wei, Socher, Richard, Li, Li-Jia, Li, Kai, and Fei-Fei, Li. Imagenet: A large-scale hierarchical image database. In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, pp. 248–255. IEEE, 2009.

Denton, Emily, Chintala, Soumith, Szlam, Arthur, and Fergus, Rob. Deep generative image models using a laplacian pyramid of adversarial networks. arXiv preprint arXiv:1506.05751, 2015.

Dosovitskiy, Alexey, Springenberg, Jost Tobias, and Brox, Thomas. Learning to generate chairs with convolutional neural networks. arXiv preprint arXiv:1411.5928, 2014.

Dosovitskiy, Alexey, Fischer, Philipp, Springenberg, Jost Tobias, Riedmiller, Martin, and Brox, Thomas. Discriminative unsupervised feature learning with exemplar convolutional neural networks. In Pattern Analysis and Machine Intelligence, IEEE Transactions on, volume 99. IEEE, 2015.

Efros, Alexei, Leung, Thomas K, et al. Texture synthesis by non-parametric sampling. In Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on, volume 2, pp. 1033–1038. IEEE, 1999.

Freeman, William T, Jones, Thouis R, and Pasztor, Egon C. Example-based super-resolution. Computer Graphics and Applications, IEEE, 22(2):56–65, 2002.

Goodfellow, Ian J, Warde-Farley, David, Mirza, Mehdi, Courville, Aaron, and Bengio, Yoshua. Maxout networks. arXiv preprint arXiv:1302.4389, 2013.

Goodfellow, Ian J., Pouget-Abadie, Jean, Mirza, Mehdi, Xu, Bing, Warde-Farley, David, Ozair, Sherjil, Courville, Aaron C., and Bengio, Yoshua. Generative adversarial nets. NIPS, 2014.

Gregor, Karol, Danihelka, Ivo, Graves, Alex, and Wierstra, Daan. Draw: A recurrent neural network for image generation. arXiv preprint arXiv:1502.04623, 2015.

Hardt, Moritz, Recht, Benjamin, and Singer, Yoram. Train faster, generalize better: Stability of stochastic gradient descent. arXiv preprint arXiv:1509.01240, 2015.

Hauberg, Sren, Freifeld, Oren, Larsen, Anders Boesen Lindbo, Fisher III, John W., and Hansen, Lars Kair. Dreaming more data: Class-dependent distributions over diffeomorphisms for learned data augmentation. arXiv preprint arXiv:1510.02795, 2015.

Hays, James and Efros, Alexei A. Scene completion using millions of photographs. ACM Transactions on Graphics (TOG), 26(3):4, 2007.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.

Kingma, Diederik P and Ba, Jimmy Lei. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

Kingma, Diederik P and Welling, Max. Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114, 2013.

Lee, Honglak, Grosse, Roger, Ranganath, Rajesh, and Ng, Andrew Y. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In Proceedings of the 26th Annual International Conference on Machine Learning, pp. 609–616. ACM, 2009.

Maas, Andrew L, Hannun, Awni Y, and Ng, Andrew Y. Rectifier nonlinearities improve neural network acoustic models. In Proc. ICML, volume 30, 2013.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th International Conference on Machine Learning (ICML-10), pp. 807–814, 2010.

Radford, Alec, Metz, Luke, & Chintala, Soumith. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv [Cs.LG]. Retrieved from http://arxiv.org/abs/1511.06434.

Rasmus, Antti, Valpola, Harri, Honkala, Mikko, Berglund, Mathias, and Raiko, Tapani. Semisupervised learning with ladder network. arXiv preprint arXiv:1507.02672, 2015.

Springenberg, Jost Tobias, Dosovitskiy, Alexey, Brox, Thomas, and Riedmiller, Martin. Striving for simplicity: The all convolutional net. arXiv preprint arXiv:1412.6806, 2014.

Vincent, Pascal, Larochelle, Hugo, Lajoie, Isabelle, Bengio, Yoshua, and Manzagol, Pierre-Antoine. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. The Journal of Machine Learning Research, 11:3371–3408, 2010.

Xu, Bing, Wang, Naiyan, Chen, Tianqi, and Li, Mu. Empirical evaluation of rectified activations in convolutional network. arXiv preprint arXiv:1505.00853, 2015.

Yu, Fisher, Zhang, Yinda, Song, Shuran, Seff, Ari, and Xiao, Jianxiong. Construction of a large-scale image dataset using deep learning with humans in the loop. arXiv preprint arXiv:1506.03365, 2015.

Zhao, Junbo, Mathieu, Michael, Goroshin, Ross, and Lecun, Yann. Stacked what-where autoencoders. arXiv preprint arXiv:1506.02351, 2015.